# An efficient, IP based Solution to the 'Logical Timestamp Wrapping' Problem


Christopher Krügel
chris@infosys.tuwien.ac.at
Distributed Systems Group
Technical University Vienna

Thomas Toth
ttoth@infosys.tuwien.ac.at
Distributed Systems Group
Technical University Vienna



## Abstract

*The order of events occurring at different nodes of a network can be critical in distributed systems [2]. As the synchronization of physical clocks with an acceptable accuracy is often infeasible in large networks, Lamport [5] introduced logical timestamps for message based systems. A prerequisite for ordering events is the encapsulation of timestamps into every sent message (i.e. timestamp wrapping).*

*We present an implementation which integrates a logical timestamp into IP packets without increasing the packet size. The difficulties of maintaining a consistent view between communication partners and a solution for this problem in case of lost, delayed or duplicated packets is presented.*


## 1 Introduction

Complex Event Processing [9] is an innovative technology to extract and analyze information from message based distributed systems which has been developed at Stanford University as part of the Rapide Project [6]. Their approach builds on the concepts of causal event modelling, event patterns and pattern matching. The technology can be applied to a wide variety of problems, ranging from middleware management to network monitoring [7] and network intrusion detection [8].

Similar to Lamports happened-before relation [5] the occurrence of local activities and the dispatch and receipt of messages constitute events. Obviously, an event is potentially causal related to another when both occur at the same node. Additionally, the receipt of a message is related to the appropriate send event. Complex event processing allows the user to model and specify a pattern of causal events. The complex event processor then attempts to find (i.e. match) such given patterns in the event data which is produced by the running system.

The Stanford Event Processor performs the collection of and computation on event data at a central location. In our opinion, this architecture reduces the fault tolerance and scalability of the approach. Especially in security and performance critical fields like intrusion detection, a distributed approach seems to be more appropriate. Therefore we are currently developing an intrusion detection system which performs event correlation in a fully distributed manner (Quicksand [3] and Sparta [4]). When events are not immediately collected (and ordered) at a central location the following problem arises. Whenever a node A sends a message to another node B that node B might want to know which events happened at A before it has sent its message. This task is easy when only one message is exchanged but becomes more complicated when several are involved. Especially when dealing with real world networks, packets can get lost, delayed or duplicated. This prevents nodes from simply counting the number of received packets and using this number to exactly specify the packet at the sending host.

In order to solve this problem, it is necessary to unambiguously identify the time when node A has sent a packet to node B (i.e. to determine the logical timestamp when A dispatched its message to B). One has to somehow mark packets with a unique identifier. We call the problem of including such an identification or a logical timestamp into each packet the *timestamp wrapping problem*.

The contribution of this paper is the evaluation of various approaches to solve the timestamp wrapping problem and the presentation of an efficient IP based solution.

## 2 Design Issues

This paper does not describe a generic method or algorithm but the actual instance (implementation) of an existing solution (i.e. logical timestamps). Therefore we had to make several design decisions for our implementation which are discussed below.

These design decisions were guided by four main requirements which our solution should fulfill.

1. *Minimal Overhead* - We wanted to impose as little overhead as possible on the participating computer systems. That means that the additional bandwidth and computing resources consumed should be minimized. As each sent packet has to contain an identification, it is especially important to keep it short to prevent unacceptable additional network load.

2. *Compatibility* - The solution should be compatible to existing network protocols. It makes no sense to create one's own protocol that encapsulates or substitutes established standard message formats. Existing nodes without timestamp support should still be able to communicate with nodes marking their packets.

3. *Fault Tolerance* - Real networks may fail and one has to deal with lost, delay or duplicated packets in current datagram (i.e. IP) based networks. We need to include enough redundancy to cope with a reasonable amount of network faults. Obviously, this requirement conflicts with the aim to achieve minimal overhead.

4. *Sufficient Granularity* - It might not be necessary for an application to uniquely identify each sent message. One could also group a few messages together and assign an unique identifier to the whole set. This would lead to less overhead but also negatively affects accuracy. To be safe (and not to miss actual causal relations) one has to to assume that all packets of the group have been sent at the same time as the last one. Thereby some incorrect causal relationships might be introduced at the sending host. Our solution has to provide sufficient granularity for a broad range of applications and optimally identifies each single packet.

To reach these partially contradicting goals, we had to make choices for a suitable protocol layer, for the exact format of the identifier and its integration into packets as well as decide between absolute and relative timestamps. For each point, a number of alternatives are described in the following subsections and the rationales for our decisions are provided.

### 2.1 Choice of Protocol Layer

The first design issue dealt with the choice of the appropriate protocol layer (according to the OSI reference model). We had to choose a layer whose messages should be marked and considered the data link (more specific ethernet), the network (IP) and the transport layer (UDP, TCP). The data link layer does not seem appropriate because it only provides a mean for higher layer packets to hop between directly connected computers on their route to the target. As our identifiers are only relevant between the actual endpoints of the message and the sender cannot influence the fragmentation performed by the data link layer between itself and the receiver it is not necessary to distinguish (and mark) packets at this level.

A look at the transport layer reveals that two different protocols, UDP and TCP are used, which had to be supported. While TCP uses its own 32 bit sequence number to tag packets, UDP carries no such information. One would have to add additional information to a UDP packet which increases bandwidth usage.

Additionally, TCP is not a packet oriented protocol but provides a reliable data stream. Such a stream does not provide enough granularity. When a session between two computers is established a lot of messages may be exchanged over such a channel. It is not enough to record the begin of the session as events may occur later and cause messages to be sent over the channel which are related to the occurrence of events at the receiving node.

Therefore it seemed natural to consider each IP packet as a message unit and create an unambiguous relation between the dispatch and the receipt of each IP packet. Whenever a node monitors interesting events as a result of a received IP packet it can easily query the sending node for interesting events which might have occurred before the message was sent. In the following discussion the terms IP packet and message are used interchangeably. The IPv4 packet header (RFC 791) is shown below.

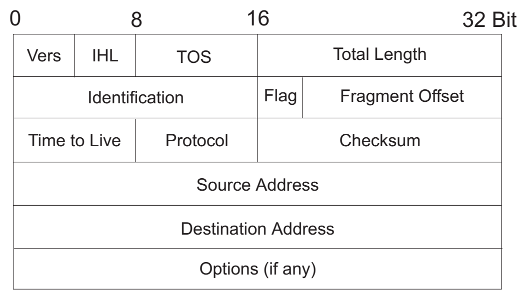


**Figure 1.** IPv4 Header

### 2.2 Timestamp Format

After the decision to mark IP packets had been made it was necessary to choose a concrete timestamp format and its integration into into the IP packets. Two possibilities were considered.

First, one could use the information available in the header itself. The first obvious choice is the 16 bit identification field that is used to reassemble fragmented packets at the target. It is simply incremented by one each time a packet is sent. Additionally, we considered the length, fragment offset and checksum field. Although the length field is another 16 bits long, most packets are rather short (to still fit into the longest ethernet frame). Therefore one would observe many identical values which is unsuitable for a unique identification. The checksum field has to be recalculated each hop when the time-to-live field is decremented and cannot be used either. One could use the 13 bit of the fragment offset when the DF (don't fragment) bit is simultaneously set. This causes packets to be dropped instead of fragmented when they do not fit into a single data link layer packet and therefore the fragment offset remains untouched. Unfortunately, the compatibility would be effected negatively as packets can not be sent over networks with a small MTU (maximum transferable unit) size.

Second, IP offers the possibility to include up to 40 bytes in an optional section. The timestamp could be added to the header inside the option field but this would use up network bandwidth proportional to the size of the identifier.

We chose not to abuse or modify the header information and only use the 2 byte identifier field. Additionally, we do not extend the size of the packets in any way by adding options to the header or data to the payload field.

### 2.3 Absolute vs. Relative Timestamp

An important decision has to be made about the design of the timestamp itself. It is necessary to uniquely identify a packet over a long period of time, optimally over the whole lifetime of the node which sends messages. Two options are available to achieve this goal.

One uses a single counter (i.e. absolute timestamp) for the complete lifetime of the node that is incremented each time a packet is sent. The counter is saved in non-volatile memory and restored on boot up so it is necessary to provide enough room to cover all packets sent by that node. In order to prevent an overflow of the counter, at least 6 bytes are needed for a heavy accessed server on a fast ethernet (and maybe more when faster networks emerge - e.g. gigabit ethernet). When the size of the IP header which is 20 bytes (see Figure 1) is compared to the size of the timestamp the bandwidth overhead is significant. Especially when one takes into account that the new version of IP (IPv6 - RFC 2460) reduces the non-address part of the header from 12 to 8 bytes.

The other option uses a shorter counter that uniquely identifies packets in a given context. The context is established implicitly or by special synchronisation packets. These synchronisation packets are acknowledged messages which negotiate a common, agreed context between the sending and receiving node. This allows to use shorter timestamps for intermediate packets. A shorter timestamp can be integrated into the IP header more easily and has the advantage of reducing the bandwidth overhead. On the contrary, the synchronization might cause additional packets to be sent and it is not possible to rely on implicit synchronization alone because of potential network faults.

A shorter timestamp causes less overhead for each individual packet but makes synchronization necessary more often. We think that an overhead of 6 to 8 bytes per packet is not acceptable and use a relative timestamp of 2 bytes. This allows us an easy integration into the IP header but demands some work to maintain a consistent context state between nodes. The way to achieve such a common context is described in the next Section 3.

## 3 Design Specification

All nodes that are attached to the network (e.g. computers on the Internet) can be divided into two sets. One set is called *active* and consists of all nodes that support the dispatch and receipt of timestamps while members of the other, called *passive*, have no notion of logical time. All active nodes send marked packets all the time and do not adjust their messages according to the receiver. That implies that marked packets have to be understood by passive nodes and guarantees compatibility to existing network protocols. Although active nodes can easily communicate with passive nodes and vice versa they have to store some state information when talking to active nodes. Therefore it is necessary for each active node to know whether its communication partner is active or not.

### 3.1 Setup Phase

The knowledge of all active nodes within a network can be disseminated in two ways. One uses a (potential hierarchical and/or distributed) name service where each active node can register itself and query for other active nodes. This would be a database that stores a list of IP addresses that support the timestamp service. While this could make sense for isolated regions with a dedicated name server the interoperablitiy between different regions is limited. For a global name service comparable to DNS the administrative overhead is definitely unacceptable. The other way which we have implemented uses a setup phase between two nodes before they communicate for the first time ever. The sending computer transmits a setup request to a well-known port at the receiving machine and waits for a response. Meanwhile the process that queued the original message for sending is suspended. When no setup reply is received the node is considered to be passive, otherwise it is definitely an active node and a context can be established. The results of such setup requests are stored in a cache and passive computers are asked again (by sending additional setup requests) after some time.

### 3.2 Timestamp Protocol

In order to uniquely mark all packets that are transmitted to each receiver the sender has to store a logical timestamp for each target. In our design, a logical timestamp is 8 bytes long and consists of two parts, a 6 byte context and a 2 byte identifier. For each packet which is sent to a node the corresponding timestamp has to be updated. A different logical timestamp is used between each pair of sender/receiver nodes (when both are active nodes). Notice that a pair of nodes that exchange messages in both directions actually uses two timestamps.

As described in Section 2.3 each packet effectively only carries the 2 byte identifier (stored in the identification field of the IP header as a relative timestamp) while the 6 byte context is established by different means (as described below). The $2^{16}$ (0 to 65535) possible identifiers defined by the 2 byte identification field are divided into three groups, called *low* and *high* and *neutral*. The low group ranges from the numbers 0 to 1024 while the high group ranges from $2^{15}$ to $2^{15} + 1024$ (32768 to 33792). The remaining numbers belong to the neutral group.

The timestamp is initialized only when two active nodes establish a connection for their first time. Even when one of both temporary becomes passive or unavailable, the context and identifier values are saved and restored the next time the other node starts its timestamp service (i.e. gets active) again. That means that timestamp values have to survive crashes and need to be kept (or mirrored) in non-volatile memory. When context information is lost due to a crash the appropriate communication partner may use setup packets to bring its partner into a consistent state again.

The context at the sender is increased by one every time the used identifiers change their group membership from neutral to low or from neutral to high. Whenever a sender marks a message with an identifier from the low group after identifiers from the high group have previously been used, the context counter is incremented. The same is true for the dispatch of the first high group identifier after previous low group ones. That means that the context counter is incremented twice during a single run-through of the identification counter. The way the protocol works for the sender is shown in Figure 2 below. Whenever a packet is ready to be sent, a state transition is made and the actions at the target state are executed. The action `record` describes the insertion of the current identifier value into the identification field of the IP header and the storage of the send event in an appropriate database.

While it is easy to assign the correct context counter value to each packet at the sender the situation is not as trivial for the receiver. The update of the context counter at the receiver can be done implicitly or explicitly by special synchronization (sync) packets. An implicit update can be done

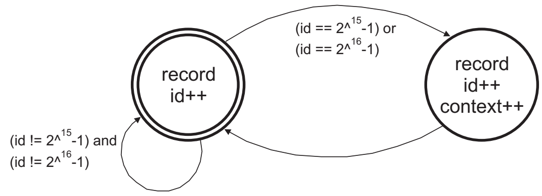


**Figure 2.** Protocol at sending Node

as soon as a low group timestamp arrives after at least one high group timestamp has been received previously and vice versa. The explicit variant is the sending of synchronization packets. Before the sender transmits its first message with a low group timestamp after high group identifiers have previously been used or vice versa, a special packet including the new context value is sent. This packet is retransmitted periodically until it has been acknowledged by the receiving host. When a number of retransmissions fail but the host is still available, it is assumed that the timestamp service has failed or is no longer active and the host becomes passive. While the synchronization process is running all new messages to the receiving host are held back. The way the protocol operates at the receiving node is shown in Figure 3 below. Whenever a packet is received, the identification field of its IP header is extracted and a state transition made. Then the actions of the target state are performed. The action `record` describes the assignment of the correct context value to the packet and the storage of the receive event in an appropriate database.

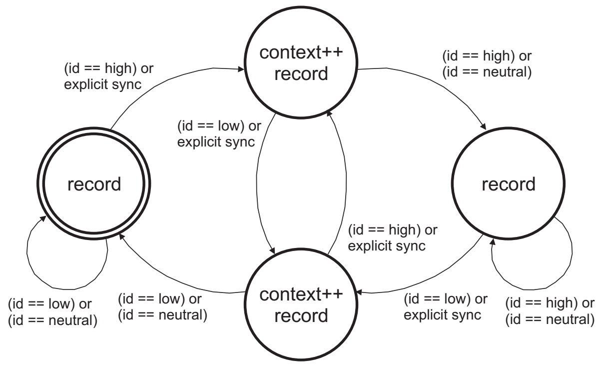


**Figure 3.** Protocol at receiving Node

Timestamps are assigned to messages at the receiving node by appending the correct context value to the packet's identification value in the following way. When a low or high group packet is received, the current context value is used. When a neutral packet with an identification field less than $2^{15}$ arrives, the largest even number that is less or equal to the current context counter value is used as its context value (e.g. 6 for a context value of 6 and 7). Packets with an identification field greater than $2^{15}$ get the largest odd number that is less or equal to the current context value (e.g. 7 for a context value of 7 and 8). This assures that delayed neutral packets are still assigned consistent context values.

### 3.3 Assumptions

In order for the implementation to work correctly the following assumptions must hold. We assume that the communication subsystem may drop, delay or duplicate packets.

- No more than 1024 consecutive packets may be dropped by the communication subsystem. Otherwise all low or high group packets might get lost and an implicit synchronization fails. When synchronization packets are used every time the context changes, this assumption can be dropped.

- A packet with timestamp n must arrive earlier than all packets with timestamps greater than $n + (2^{15} - 1024)$ or not arrive at all. This is necessary for neutral group packets to be assigned to the correct context. Additionally, a low group packet that is delayed until later high group packets are received causes an illegal increment of the context counter. The same is obviously true for delayed high group packets.

  While the incorrect increment of the context can be prevented by always sending synchronization packets, the incorrect assignment of packets to wrong contexts cannot. Unfortunately, this problem is always existent when utilizing relative counters and allowing arbitrary delays of packets.

  The sending of a synchronization packets can help when packets are regularly delayed. As new packets are held back during the synchronization process the time it may take old packets to arrive is extended by at least the round trip time needed for the sync messages. This reduces the likelihood of incorrect assigned packets.

As stated above the loss of packets during transmission cause no problems as long as they occur infrequent enough or sync packets are used. The duplication of packets pose no problems either as the receiver simply records the arrival of two or more packets with the same timestamp in its event log. The pattern matching process itself will have to deal with such duplicates but as the same packet has been received multiple times it has to be recorded that often. The most severe problem are delayed packets which are assigned to a wrong context or which cause premature (i.e. illegal) context increments at the receiver.

The following paragraph deals with consequences of inconsistent context updates or incorrectly assigned packets. Whenever a packet is assigned to a wrong context (assuming the context values are consistent) it gets a later timestamp than it actually should. When the pattern matching algorithm eventually uses that timestamp to query the sending host for events that have happened-before, this might result in the detection of actually non-existing relationships. In our area of application (i.e. intrusion detection), the consequence could be a false positive, the raise of an unjustified alarm. Nevertheless, the occurrence of an incorrect warning is much less severe than the opposite, the miss of an actual intrusion. Therefore, a rare incorrect assignment of a packet to a later context can be tolerated.

Inconsistent updates of the context counter may occur in two different ways.

On one hand, the loss of all low or high group packets results in a smaller context counter value at the receiver. When such an inconsistency is detected during the exchange of sync packets, the context parts of all events received since the last successful synchronization are increased accordingly and the context value is adjusted correctly. This causes all affected messages to be potentially dated later than they have actually been sent. This might result in the detection of non-existing relations as some (or even all events) are dated later than they actually have occurred. But as stated previously, to be on the safe side events are assumed to have happened at the latest possible point in time.

On the other hand, a delayed low or high group packet might result in a larger context counter value at the receiving node. In such a situation, the packets might only have been dated later. As stated above, this situation can be considered acceptable and so only the context value is adjusted.

### 3.4 Adaptive Synchronization

In order to decrease the network load we use a technique that we call *adaptive synchronization*. This approach starts by having the sending host transmitting synchronization packets after every second context change. When the connection between both nodes is reliable the sending host notices that all synchronization packets are immediately acknowledged and both nodes have a consistent view of the current context. This allows to gradually relax the dispatch of sync packets and rely upon implicit synchronization. Whenever a sync packet needs retransmission or the context values do not match, the time between explicit synchronization is immediately reduced.

### 3.5 Outlook on future IP versions

Another interesting question is the applicability of our approach for the new version of IP, namely IPv6. As stated above the size of the non-address part of the IPv6 header has been reduced to 12 bytes. The field for identifying a packet in case of fragmentation was moved to the optional header part. Nevertheless, a 20 bit flow label field was introduced which is intended to uniquely mark a sequence of packets from a source to one or multiple targets. It seems naturally to use this field as a new, slightly extended identification field. This would allow us to reduce the number of necessary explicity synchronization packets by a factor of 16.

# 4 Implementation

Our implementation is developed in C under Linux 2.2. We needed to be able to do the following things.

First, our module has to send and receive synchronization messages (realized as UDP packets). This can easily be done with normal privileges in user mode by using the network (socket) interface.

Second, we must be able to inspect outgoing as well as incoming IP packets. This is necessary to read and set the identification field of the IP packet headers of messages that are exchanged with other active hosts. Reading could either be done by using a packet filter interface like BPF (BSD packet filter) as used by `libpcap` or by writing a kernel module that uses the firewall interface. The writing of IP header fields requires a kernel module with direct access to the packet or a kernel patch.

Third, packets have to be delayed during the synchronization process. Whenever a process attempts to send a packet to a target host where currently sync packets are exchanged, it is put to sleep. This can be realized directly inside the kernel (network code) or by a kernel firewall module.

The implementation is realized as a combination of a daemon process and a kernel module using the firewall support (compiled into the kernel). The kernel module is responsible for updating and reading the identification fields while the daemon process runs in user space and performs the synchronization. The communication between both parts is done via system calls.

## 4.1 Performance

This subsection deals with the performance impact of our logical timestamp implementation. We generate additional overhead at two places.

One is the dispatch and receipt of sync packets. This happens at most once every $2^{15}$ packets and when adaptive synchronization is used considerably less often. The size of a UDP packet inclusive all header information (20 bytes for IP, 8 bytes for UDP) and payload (maximal 12 bytes in our implementation) is limited to 40 bytes. When two packets are exchanged the resulting proportional increase in bandwidth usage is 0.002 bytes per packet. Each regular IP packet is sent without any additional overhead and the network bandwidth remains unaffected.

The other is the processing overhead at the communication partners. Whenever a packet should be sent to an active host the appropriate identifier has to be loaded and written to the IP header. A hashtable is used to map IP address of target hosts to their current context and identifier values. Additionally, the dispatch and the receipt of a packet has to be stored in the event database for later analyzation. Compared with the normal processing of incoming and outgoing packets this effort is nevertheless negligible.

## 4.2 Storage Considerations

It is necessary for a host to store the logical timestamps (8 bytes) for each active host that it is communicating with. The timestamps have to be saved between reboots and it is necessary to have an entry for every active node that host has exchanged messages with. Parts of this database are held on disk and only entries that are currently used are loaded into kernel memory space into a hashtable. Even when assuming that a host contacts a few thousand active nodes the size of the data remains relatively small (a few kilobytes) which seems acceptable for current systems.

A more severe problem is the storage of the event data itself. The granularity of using IP packets is very fine and a node that sends and receives packets at full network speed can create thousands of events (i.e. packets) every second. It is obvious that not every packet should be recorded in an event log. We aggregate all event information produced by packets between interesting external events (events not caused by dispatch or receipt of an IP packet) into intervals. That means that only a single entry in the event database has to be for all packets that are received between two external events.

# 5 Applications

Every application that needs to relate event data from different distributed nodes might profit from our approach. We assume that events at a node B can only be influenced by events from node A when a message has been sent from node A to B after the relevant event occurred at node A and before the relevant event took place at node B. Current systems that use such assumptions usually gather event data at a central location from different nodes. This simplifies the ordering of processing of events as information can immediately be time stamped on arrival and is available locally. Most systems that correlate events from different sources to deduce additional information about the network state follow this approach. This ranges from network management tools like HP Open View [12] to intrusion detection (ID) systems. Advanced research ID systems like Emerald [10] or GrIDS [13] as well as commercial ones like RealSecure [11] use this technique. Unfortunately this method suffers from the well-known drawbacks of client/server architectures. All event data has to be sent to a central location causing tremendous network traffic. In addition to that, limited scalability and fault tolerance are also issues.

These systems ease such drawbacks by introducing server hierarchies, filters and redundancy but this merely fights the symptoms. We think that these limitations can be circumvented with a distributed approach that stores event data locally and detects patterns in a fully distributed fashion [3].

One difficulty when following such a distributed approach is the problem of determining at a receiver node which events have happened at the sender before it dispatched a certain message, especially after a longer period of time when clock drift effects or resets of physical clocks are issues. One possibility is the introduction of a service that synchronizes the physical clocks with an acceptable accuracy and use physical timestamps to mark the sending and receipt of events. This is a variant that seems less than desirable in large, heterogenous network environments and again introduces a central service into the system. The other variant uniquely marks packets and allows a receiver to refer to such an identifier to inform the sender which packet it is interested in.

Every system that wants to perform a distributed pattern matching needs to determine the events that happened at the sender node before it transmitted a certain message. This can be done in an efficient way with logical timestamps.

## 6 Conclusion

We have presented an universal IP based approach to efficiently exchange timestamp information in a message based distributed systems. Our primary design goal was the minimization of the introduced overhead. As network traffic between nodes can be very high, only a lightweight solution was acceptable. We tried to achieve this by adding no additional information to each regular packet and perform explicit synchronization in an adaptive manner as rarely as possible.

Our implementation could be applied for numerous systems which need to know the order of distributed events. Ranging from our primary application, an intrusion detection system (see Quicksand [3] and Sparta [4]) to other network management tools it could also applied to replication services like ISIS [1].

An additional goal is the realization of adequate fault tolerance. The network layer of the Internet is infamous for coming up with unpleasant surprises and to cite Tanenbaum [14]:

> It (the packet) may bounce around for a while, go into hiding in a far corner of the world and the suddenly emerge at an inopportune moment 30 sec later.

We do not want to have a single packet that violates one of our assumptions renders our protocol completely useless. Therefore inconsistencies are eventually detected and previously collected data mapped to a safe state.

## References


[1] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. Computer Systems*, 5(1):47–76, 1987.

[2] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, Harlow, England, 2nd edition, 1996.

[3] C. Krügel and T. Toth. Quicksand - a fully distributed intrusion detection system. In *planned submission to RAID 2001*, 2001.

[4] C. Krügel and T. Toth. Sparta - a security policy reinforcement tool for large networks. In *submitted to USENIX Security Symposium*, 2001.

[5] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.

[6] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV*, Princeton University, July 1996.

[7] NetView. Network monitoring tool. http://www.stanford.edu/~dbrumley/NetView/.

[8] L. Perrochon, E. Jang, and D. C. Luckham. Enlisting event patterns for cyber battlefield awareness. In *ARPA Information Survivability Conference and Exposition (DISCEX'00)*. IEEE Computer Society Press, 2000.

[9] L. Perrochon, S. Kasriel, and D. C. Luckham. Managing event processing networks. Technical Report CSL-TR-99-877, Stanford Computer Systems Laboratory, October 1999.

[10] P. A. Porras and P. G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, October 1997.

[11] Realsecure. http://www.iss.net/customer_care/ resource_center/product_lit/.

[12] K. R. Sheers. Hp openview event correlation. *Hewlett-Packard Journal*, October 1996.

[13] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids - a graph based intrusion detection system for large networks. In *Proceedings of the 20th National Information Systems Security Conference*, volume 1, pages 361–370, October 1996.

[14] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2nd edition, 1989.